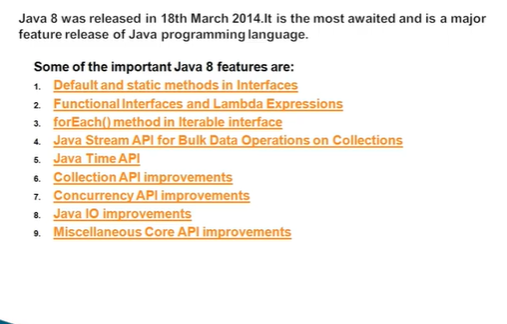
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| =   |  | | --- | | Java SE 8 | |  | March 18, 2014 |  | JSR 335, JEP 126: Language-level support for lambda expressions (officially, lambda expressions; unofficially, closures) under Project Lambda[243] and default methods (virtual extension methods)[244][245][246] which allow the addition of methods to interfaces without breaking existing implementations. There was an ongoing debate in the Java community on whether to add support for lambda expressions.[247][248] Sun later declared that lambda expressions would be included in Java and asked for community input to refine the feature.[249] Supporting lambda expressions also enables functional-style operations on streams of elements, such as MapReduce-inspired transformations on collections. Default methods allow an author of an API to add new methods to an interface without breaking the old code using it. Although it was not their primary intent,[244] default methods also allow multiple inheritance of behavior (but not state). JSR 223, JEP 174: Project Nashorn, a JavaScript runtime which allows developers to embed JavaScript code within applications JSR 308, JEP 104: Annotation on Java types[250] Unsigned integer arithmetic[251] JSR 337, JEP 120: Repeating annotations[252] JSR 310, JEP 150: Date and time API[253] JEP 178: Statically-linked JNI libraries[254] JEP 153: Launch JavaFX applications (direct launching of JavaFX application JARs)[255] JEP 122: Remove the permanent generation[256] |
|  |  |  |  |  |

‘

‘





# Default and static methods in Java 8 Interface\_PART1

## Why default method?

For example, if several classes such as A, B, C and D implements an interface XYZInterface then if we add a new method to the XYZInterface, we have to change the code in all the classes(A, B, C and D) that implements this interface. In this example we have only four classes that implements the interface which we want to change but imagine if there are hundreds of classes implementing an interface then it would be almost impossible to change the code in all those classes. This is why in java 8, we have a new concept “default methods”. These methods can be added to any existing interface and we do not need to implement these methods in the implementation classes mandatorily, thus we can add these default methods to existing interfaces without breaking the code.

We can say that concept of default method is introduced in java 8 to add the new methods in the existing interfaces in such a way so that they are backward compatible. Backward compatibility is adding new features without breaking the old code.

1. Java interface default methods will help us in extending interfaces without having the fear of breaking implementation classes.
2. Java interface default methods has bridge down the differences between interfaces and abstract classes.
3. Java 8 interface default methods will help us in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.
4. Java interface default methods will help us in removing base implementation classes, we can provide default implementation and the implementation classes can chose which one to override.
5. One of the major reason for introducing default methods in interfaces is to enhance the Collections API in Java 8 to support lambda expressions.
6. If any class in the hierarchy has a method with same signature, then default methods become irrelevant. A default method cannot override a method from java.lang.Object. The reasoning is very simple, it’s because Object is the base class for all the java classes. So even if we have Object class methods defined as default methods in interfaces, it will be useless because Object class method will always be used. That’s why to avoid confusion, we can’t have default methods that are overriding Object class methods.
7. Java interface default methods are also referred to as Defender Methods or Virtual extension methods.

## ****3. Default Interface Methods in Action****

### Scenario One: Default Method in Interface

To understand using default method, I am creating an interface Village that has some method declarations and one default method. Default method starts with default keyword. *By default all methods of interface will be public, so no need to use public keyword to declare and define methods in interface.*   
**Village.java**

package com.concretepage;

public interface Village {

void setNumOfPeople(int num);

void setName(String name);

default String getBusinessType(){

return "Most of the Village people do Farming";

}

}

Create a Location class that will implement Village interface. Default method will automatically be available in this class.   
**Location.java**

package com.concretepage;

public class Location implements Village {

public int noOfPeople;

public String name;

@Override

public void setNumOfPeople(int n){

this.noOfPeople = n;

}

@Override

public void setName(String name){

this.name = name;

}

}

To test the scenario, create a Main class and access default method by Location object.   
**Main.java**

package com.concretepage;

public class Main {

public static void main(String[] args){

Location lo = new Location();

System.out.println(lo.getBusinessType());

}

}

Output will be as below.

Most of the Village people do Farming

### Scenario Two: Static Method in Interface

Now we can write static method in interface too. In our Village interface, I have declared getVillageId() as an static method. This static method can be accessed in default method as well.   
**Village.java**

package com.concretepage;

public interface Village {

void setNumOfPeople(int num);

void setName(String name);

static int getVillageId(){

return 1;

}

default String getBusinessType(){

return "Business type is Farming and village id:"+getVillageId();

}

}

I am doing some changes in Location class to use static method. We can use static method by interface name.   
**Location.java**

package com.concretepage;

public class Location implements Village {

public int noOfPeople;

public String name;

@Override

public void setNumOfPeople(int n){

this.noOfPeople = n;

}

@Override

public void setName(String name){

this.name = name;

}

public int getLocationId(){

return Village.getVillageId();

}

}

Find the main method to test the static method.   
**Main.java**

package com.concretepage;

public class Main {

public static void main(String[] args){

Location lo = new Location();

System.out.println(lo.getBusinessType());

System.out.println("Village id:"+Village.getVillageId());

System.out.println("Location Id:"+lo.getLocationId());

}

}

Run the Main class and check the output.

Business type is Farming and village id:1

Village id:1

Location Id:1

### Scenario Three: Multiple Inheritance- Default Method with Same Name in Two Interfaces

In multiple inheritance scenarios, where a class implements more than one interface, we need to check how default method behaves. Now I am creating one more interface that contains getBusinessType() as default method.   
**City.java**

package com.concretepage;

public interface City {

void setName(String name);

void setArea(int area);

default String getBusinessType(){

return "Service";

}

}

For multiple inheritances, Location class will implement Village and City interfaces both. As Village and City both contains default method with same name, so because of ambiguity, the Location class will force to define that default method explicitly in the class. The Location class will not compile until we define a method with same name as default method.   
**Location.java**

package com.concretepage;

public class Location implements Village, City {

public int noOfPeople;

public String name;

public int area;

@Override

public void setNumOfPeople(int n){

this.noOfPeople = n;

}

@Override

public void setName(String name){

this.name = name;

}

@Override

public void setArea(int area){

this.area = area;

}

@Override

public String getBusinessType(){

return "People do business like Farming and Service.";

}

public int getLocationId(){

return Village.getVillageId();

}

}

Run the Main class and the output will be as below.

People do business like Farming and Service.

Village id:1

Location Id:1

## Default Method and Multiple Inheritance

The [multiple inheritance](https://beginnersbook.com/2013/05/java-multiple-inheritance/) problem can occur, when we have two interfaces with the default methods of same signature. Lets take an example.

interface MyInterface{

default void newMethod(){

System.out.println("Newly added default method");

}

void existingMethod(String str);

}

interface MyInterface2{

default void newMethod(){

System.out.println("Newly added default method");

}

void disp(String str);

}

public class Example implements MyInterface, MyInterface2{

// implementing abstract methods

public void existingMethod(String str){

System.out.println("String is: "+str);

}

public void disp(String str){

System.out.println("String is: "+str);

}

public static void main(String[] args) {

Example obj = new Example();

//calling the default method of interface

obj.newMethod();

}

}

Output:

Error: Duplicate default methods named newMethod with the parameters () and () are inherited from the types MyInterface2 and MyInterface

This is because we have the same method in both the interface and the compiler is not sure which method to be invoked.

## How conflicts are resolved while calling default methods?

So far so good. We have got all basics well. Now move to complicated things. In java, a class can implement N number of interface. Additionally, a interface can also extend another interface as well. An if any default method is declared in two such interfaces which are implemented by single class. then obviously class will get confused which method to call.

**Rules for this conflict resolution are as follows:**

**1)** Most preferred are the overridden methods in classes. They will be matched and called if found before matching anything.  
**2)** The method with the same signature in the “most specific default-providing interface” is selected. This means if class Animal implements two interfaces i.e. Moveable and Walkable such that Walkable extends Moveable. Then Walkable is here most specific interface and default method will be chosen from here if method signature is matched.  
**3)** If Moveable and Walkable are independent interfaces then a serious conflict condition happen, and compiler will complain then it is unable to decide. The you have to help compiler by providing extra info that from which interface the default method should be called. e.g.

|  |
| --- |
| Walkable.super.move();  //or  Moveable.super.move(); |

PROCEDURE 2

For multiple inheritances, Location class will implement Village and City interfaces both. As Village and City both contains default method with same name, so because of ambiguity, the Location class will force to define that default method explicitly in the class. The Location class will not compile until we define a method with same name as default method.   
**Location.java**

package com.concretepage;

public class Location implements Village, City {

public int noOfPeople;

public String name;

public int area;

@Override

public void setNumOfPeople(int n){

this.noOfPeople = n;

}

@Override

public void setName(String name){

this.name = name;

}

@Override

public void setArea(int area){

this.area = area;

}

@Override

public String getBusinessType(){

return "People do business like Farming and Service.";

}

public int getLocationId(){

return Village.getVillageId();

}

}

Run the Main class and the output will be as below.

People do business like Farming and Service.

Village id:1

Location Id:1

## ****5. Static Interface Methods****

Aside from being able to declare default methods in interfaces, **Java 8 allows us to define and implement staticmethods in interfaces**.

Since static methods don’t belong to a particular object, they are not part of the API of the classes implementing the interface, and they have to be **called by using the interface name preceding the method name**.

To understand how static methods work in interfaces, let’s refactor the Vehicle interface and add to it a static utility method:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public interface Vehicle {        // regular / default interface methods        static int getHorsePower(int rpm, int torque) {          return (rpm \* torque) / 5252;      }  } |

**Defining a static method within an interface is identical to defining one in a class.** Moreover, a static method can be invoked within other static and default methods.

Now, say that we want to calculate the [horsepower](https://en.wikipedia.org/wiki/Horsepower) of a given vehicle’s engine. We just call the getHorsePower()method:

|  |  |
| --- | --- |
| 1 | Vehicle.getHorsePower(2500, 480)); |

The idea behind static interface methods is to provide a simple mechanism that allows us to **increase the degree of**[**cohesion**](https://en.wikipedia.org/wiki/Cohesion_(computer_science)) of a design by putting together related methods in one single place without having to create an object.

Pretty much **the same can be done with abstract classes.** The main difference lies in the fact that **abstract classes can have constructors, state, and behavior**.

Furthermore, static methods in interfaces make possible to group related utility methods, without having to create artificial utility classes that are simply placeholders for static methods.

## Java 8 Example: Static method in Interface

As mentioned above, the static methods in interface are similar to default method so we need not to implement them in the implementation classes. We can safely add them to the existing interfaces without changing the code in the implementation classes. Since these methods are static, we cannot override them in the implementation classes.

interface MyInterface{

/\* This is a default method so we need not

\* to implement this method in the implementation

\* classes

\*/

default void newMethod(){

System.out.println("Newly added default method");

}

/\* This is a static method. Static method in interface is

\* similar to default method except that we cannot override

\* them in the implementation classes.

\* Similar to default methods, we need to implement these methods

\* in implementation classes so we can safely add them to the

\* existing interfaces.

\*/

static void anotherNewMethod(){

System.out.println("Newly added static method");

}

/\* Already existing public and abstract method

\* We must need to implement this method in

\* implementation classes.

\*/

void existingMethod(String str);

}

public class Example implements MyInterface{

// implementing abstract method

public void existingMethod(String str){

System.out.println("String is: "+str);

}

public static void main(String[] args) {

Example obj = new Example();

//calling the default method of interface

obj.newMethod();

//calling the static method of interface

MyInterface.anotherNewMethod();

//calling the abstract method of interface

obj.existingMethod("Java 8 is easy to learn");

}

}

Output:

Newly added default method

Newly added static method

String is: Java 8 is easy to learn